arkus Fischer

Parallelstrategie

oftware für Rechner-Cluster ntwickeln

t Linux lassen sich zwar für wenige tausend Mark stungsfähige Rechner-Cluster aufbauen, oft ngelt es aber an der passenden Software. s trifft 3D-Grafiker härter als Forscher und hniker, die ebenfalls einen gehörigen Appetit Rechenleistung verspüren. Sie schreiben e Software für Simulationen und numerische erimente meist selbst – und können sie deshalb nell auf einen Cluster umsetzen. Beim Basteln Algorithmen helfen bewährte Richtlinien. d die nötige Betriebssoftware steht zum Beispiel dem kostenlosen System PVM bereit.

Von Ian Foster [1] stammt der allgemein anerkannte 'methodische Entwurf' für Software, die auf Computern mit mehreren parallelen Rechenwerken laufen soll – ob Vektor-Supercomputer Marke Cray, Windows-Multiprozessor-PC oder Linux-Cluster. In Anlehnung an Foster kann man mit einem hardwareunabhängigen Entwurf beginnen und diesen anschließend an die jeweilige Maschine anpassen.

Den Entwurf geht man in vier Schritten an:

- Partitionierung (Zerlegung in Teilaufgaben),
- Auslegung der Kommunikation,
- Agglomeration (Bündelung von Aufgaben),
- Mapping (Aufteilung auf Prozessoren).

Die ersten beiden Schritte zielen auf Nebenläufigkeit – das eigentliche Parallelrechnen – und Skalierbarkeit, der Beschleunigung mit zunehmender Zahl an Prozessoren. Die letzten Schritte berücksichtigen die Eigenschaften der vorhandenen Hardware. Zum Beispiel legt man Prozesse, die viele Daten miteinander austauschen müssen, am besten auf denselben Prozessor.

Partitionierung: In diesem Schritt wird die Berechnung in möglichst viele Teilaufgaben zerlegt – 'granularisiert'. Man ignoriert die tatsächliche Zahl der Prozessoren, sondern geht zunächst davon aus, eine genügende Anzahl zur Verfügung zu haben. Im wesentlichen gilt es, gute Konzepte zur Parallelisierung zu verfolgen. Dies schafft das größte Potential für die parallele Ausführbarkeit; die Optimierung auf die Zielarchitektur folgt später.

Eine gute Partitionierung unterteilt sowohl die Rechenschritte (Functional Decomposition) als auch die Daten (Domain Decomposition). Ein Beispiel für das erstere wäre, zur Klimavorhersage die Ozeane von einem anderen Prozessor bearbeiten zu lassen als die Atmosphäre; auch die Daten könnte man hier unterteilen, beispielsweise in Nord- und Südhalbkugel. Um den Kommunikationsaufwand zu verkleinern, kann es sinnvoll sein, identische Daten bei mehreren Prozessen zu speichern (Replikation).

Auslegung der Kommunikation: Die Teilaufgaben tauschen Daten miteinander aus – im Beispiel wäre das etwa die Temperatur der Meeresoberfläche oder die Luftbewegungen am Äquator. In der zweiten Phase untersucht man den Kommunikationsaufwand und versucht, effiziente Verfahren dafür zu finden. Beim einfachen Master/Slave-Modell, in dem die Slave-Prozesse untereinander keine Nachrichten austauschen, muß lediglich untersucht werden, ob es am Master zu einem Datenstau kommen kann.

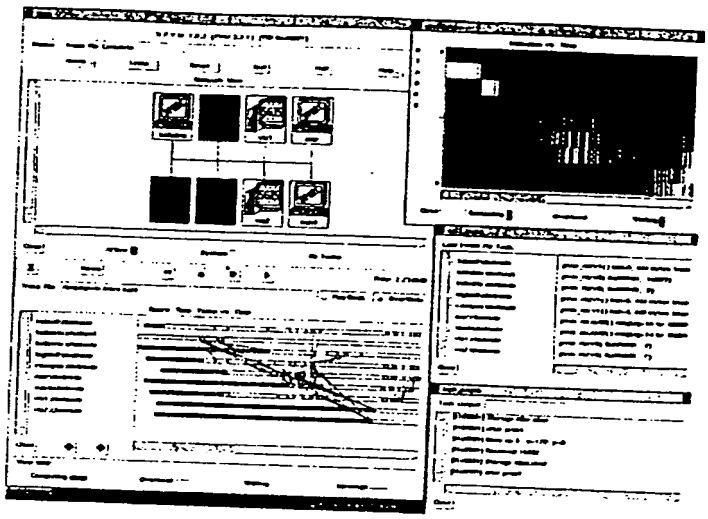
Schwieriger zu analysieren sind verteilte Anwendungen, bei denen fast jeder Prozeß mit fast jedem anderen Daten austauscht. Beispielsweise läuft ein Prozessor, der auf die Ergebnisse eines anderen wartet, nutzlos im Leerlauf. Bei komplizierteren Anwendungen kann sogar der gesamte Rechenprozeß völlig blockiert werden ('Deadlock'), etwa weil Prozeß A auf ein Ergebnis von Prozeß B wartet, B aber gleichzeitig auf ein Ergebnis von A.

Agglomeration: Nachdem es in den ersten zwei Schritten lediglich um die Parallelisierung ging, beachtet man in der dritten Phase die Leistung und die Kosten der Hardware. Erweisen sich die Ergebnisse der Zerlegung feiner, als es angesichts der verfügbaren Zahl von Prozessoren sinnvoll ist, muß man Entscheidungen aus den ersten Abschnitten revidieren; dann werden bislang getrennte Aufgaben vereint, um mit weniger Prozessoren und geringerem Kommunikationsaufwand auszukommen.

Mapping: Zuletzt wird festgelegt, welche Prozesse zusammen auf einem bestimmten Prozessor laufen sollen. Das kann ein für allemal fest vorgegeben sein oder dynamisch beim Programmablauf entschieden werden. Vollzieht die Anwendung regelmäßige Rechen- und Kommunikationsmuster und läuft zudem in einer störungsfreien Umgebung, bietet sich ein statisches Mapping an; dieses kann man sorgfältig optimieren. Viele Umgebungen erfordern jedoch ein dynamisches Mapping, zum Beispiel ein Cluster, an dessen Rechner zusätzlich lokale Benutzer arbeiten.

Leider laufen die vier Teilschritte des 'methodischen Ent-

wurfs' nicht strikt nacheinander ab. So kann es beispielsweise passieren, daß die Verteilung der Aufgaben auf Prozessoren selbst eine neue Aufgabe generiert: die dynamische Lastverteilung.



xpvm stellt Ressourcen und genutzte Rechenleistung dar. Es zeigt den zeitlichen Ablauf der Nachrichten und extrahiert Sender, Empfänger sowie Message Tags.

N ben- und miteinander

Es gibt im wesentlichen drei Konzepte, Programme mit Hilfe mehrerer CPUs zu beschleunigen: Multithreading und Shared Virtual Memory für Rechner, in denen mehrere Prozessoren auf denselben Speicher zugreifen, und Message Passing zur Verbindung durch Austausch von Nachrichten

Unterstützt das Betriebssystem den Multiprozessorbetrieb, läßt sich am einfachsten Leistung gewinnen, indem man die Aufgaben auf mehrere nebenläufige Threads verteilt – leichtgewichtige Prozesse in einem gemeinsamen Adreßraum. Threads gemäß POSIX-Standard finden sich in Linux und Windows NT. Hier entscheidet das Betriebssystem, wann welcher Prozeß auf welchem Prozessor läuft. Sinnvollerweise verwendet man mindestens so viele Threads, wie Prozessoren bereitstehen.

Die Threads haben alle Zugriff auf denselben Speicherraum (anders als eigenständige Prozesse, bei denen das erst durch Shared Memory nachgebildet werden muß). Threads können deshalb Daten schnell miteinander austauschen; sie dürfen sich aber beim Zugriff auf gemeinsam genutzte Informationen nicht in die Quere kommen: So könnte ein Thread eine Datenstruktur aktualisieren, während ein anderer sie gerade liest – und deshalb ein Gemisch aus alten und neuen Daten erhält. Solche Situationen verhindert man mit Hilfe von Semaphoren' (im ursprünglichen Wortsinn Flaggenzeichen). Idealerweise setzt der erste Thread vor dem Zugriff eine Semaphore; der Thread, der den Wert verändern will, muß so lange warten, bis der andere Thread die Semaphore wieder freigibt.

Die Grenzen der heutigen Betriebssysteme liegen bei etwa acht Prozessoren per Motherboard; aber schon Rechner mit vier Prozessoren sind extrem selten. Der Trend geht dahin, Dual-Prozessor-Rechner als Knoten miteinander zum Cluster zu vernetzen. Dazu dient das Message Passing; mit seiner Hilfe sieht der Anwendungsprogrammierer zum Beispiel nicht vier Knoten mal zwei Prozessoren, sondern acht Prozessoren, die frei Daten austauschen können: Die Kommunikation zwischen den Knoten geschieht transparent.

Bei Threads oder Shared Memory muß der Programmierer eher darauf achten, Datenfelder korrekt aufzuteilen. Dagegen hat er beim Message Passing sicherzustellen, daß jede gesendete Nachricht auch abgenommen wird. Die Fehlersuche hat es in sich; oft fehlt ein Debugger, der nicht nur lokale Threads analysieren kann, sondern auch Prozesse, die auf einem entfernten Rechner laufen.

Wesentlicher Vorteil des Message Passing ist die Skalierbarkeit. Gepackte Daten werden in Form von Nachrichten an einen Empfänger verschickt, der diese Nachricht mittels einer Empfangsroutine aufnimmt. Das klappt für zwei Prozessoren auf demselben Motherboard ebenso wie für tausend im Internet verstreute Rechner. Message Passing kann auf übliche Netzwerkschnittstellen aufbauen, aber ebenso spezielle Hardware und Software verwenden wie der Superrechner Intel Paragon. Von seiten der Betriebssoftware erfordert Message Passing allerdings deutlich mehr Programmieraufwand als etwa Shared Memory.

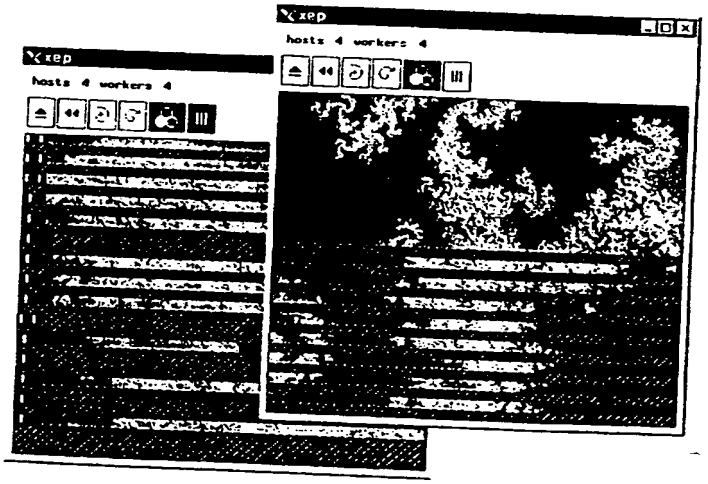
Vorgefertigt

Für Anwendungsprogrammierer stehen Bibliotheken bereit, die einen standardisierten 'Message Passing Layer' unabhängig von der Hardware anbieten. Typische Vertreter sind MPI (Message Passing Interface) und PVM (Parallel Virtual Machine). MPI eignet sich hervorragend für homogene Parallelrechner; mit 128 Befehlen bietet es eine Fülle von Kommunikations- und Gruppenfunktionen. PVM dagegen ist die bessere Wahl für 'wild' zusammengestellte Cluster. Seine 36 Funktionen erlauben, die Umgebung zur Laufzeit dynamisch zu gestalten. Man kann nicht nur Prozesse hinzufügen und entfernen, sondern sogar komplette Rechner – in puncto Fehlertoleranz und Handhabbarkeit ein entscheidender Vorteil.

PVM will dem Benutzer eine möglichst transparente, dynamisch frei konfigurierbare Programmierumgebung bieten [2]. Ein Message-Passing-Algorithmus kann, für die jeweilige Hardware kompiliert, parallel auf der sogenannten virtuellen Maschine ausgeführt werden. Auf den Einzelrechnern muß dabei nicht überall dasselbe Betriebssystem laufen: PVM ist für nahezu alle Unix-Derivate und für 32-Bit-Windows verfügbar. [3] Viele Linux-Distributionen, darunter S.U.S.E., enthalten die Software.

PVM-Dämonen kontrollieren die virtuelle Maschine. Sie laufen dauerhaft auf jedem Knoten, verwalten die dortigen Prozesse und stellen die Verbindungen her. Prozesse (Tasks) erhalten vom lokalen PVM-Dämon eine im Netz eindeutige Identifikationsnummer (ID); sie können Nachrichten an andere ID-Nummern direkt verschicken oder vom Dämon weiterleiten lassen.

Zwar führt das dynamische Konzept zu Leistungseinbußen. Aber erst in speziellen Hochgeschwindigkeitsnetzen (siehe Kasten auf Seite 150) erzielt PVM beim Aufbau und beim Durchsatz der Verbindungen deutlich schlechtere Resultate als MPI. Leistung um jeden Preis ist aber gar nicht der Sinn



Teil des PVM-Pakets ist die grafische Version xep der Apfelmännchen-Berechnung. Sie zeigt, wie einzelne Streifen schon berechnet sind, auf andere Ergebnisse aber noch gewartet wird.

von PVM, sondern beispielsweise die unproblematische Kombination von Hardware aus verschiedenen 'Lagern': So konvertiert es die Byte-Reihenfolge (little endian/big endian), um Nachrichten zwischen verschiedenen Rechnerplattformen korrekt auszutauschen.

PVM bietet ein individuell gestaltbares Programmierumfeld für Wissenschaft, Forschung und Praxis. An seine Software-Schnittstellen lassen sich zum Beispiel eigene Lastverteilungsalgorithmen andocken. Anwender haben auch ein Checkpointing für PVM entwickelt: damit läßt sich ein Zwischenstand der Rechnung auf der Festplatte sichern, um später die Rechnung fortzuführen. Das rettet die Ergebnisse der Arbeit einer Woche, wenn das System abstürzt. Außerdem wird Prozeßmigration möglich: Ein eingefrorener Prozeß kann auch auf einem anderen Rechner weiterlaufen – zum Beispiel, weil der ursprüngliche Rechner ausgefallen ist.

PVM erlaubt, von zentraler Stelle aus die Prozesse zu debuggen, die auf den Rechnern des Clusters laufen. Hierzu muß lediglich bei der Startoption eines neuen Prozesses das Flag PvmTaskDebug gesetzt werden. Dadurch startet zunächst ein Debugger (z. B. xxgdb) auf dem jeweiligen Rechner. Setzt man die Variable DISPLAY vor dem Start von pvm entsprechend, holt der X-Server die Bildschirmanzeige auf den lokalen Rechner:

```
setenv DISPLAY host:0.0
setenv PVM_EXPORT DISPLAY
```

xpvm, eine grafische Anwendung, zeigt die ausgetauschten Nachrichten und die Zustände der Prozesse. Diese Analyse kostet aber Rechenzeit, weil die Knoten für jedes anzuzeigende Ereignis entsprechende Nachrichten an xpvm senden müssen.

Fraktal parallel

Als Beispiel für PVM-Programmierung finden Sie auf Seite 151 und erweitert in der c't-Mailbox sowie auf www.heise.de/ct/ftp/ ein Parallelprogramm für das Apfelmännchen. Das Listing ist mit nur wenigen Kommentaren auch ohne PVM-Handbuch verständlich – ein weiterer Pluspunkt für diese Schnittstelle. Historische Anmerkung: c't hat auch schon über eine Cluster-Version dieses Algorithmus auf Basis von Windows und DCOM berichtet. [4]

Weil jedes Pixel des Apfelmännchens getrennt von allen anderen berechnet wird, empfiehlt sich Data Decomposition: Eine zentrale Instanz zerlegt das Bild in Unterbereiche und verteilt diese auf mehrere Arbeitsprozesse. Die feinste Granularität bestünde darin, jedem Prozessor genau ein Pixel zur Berechnung zu geben. Nun verfügt aber nicht jeder über 800 × 600 Prozessoren – und außerdem wäre der zentrale Prozeß dermaßen lange mit dem Verschicken und Empfangen der Daten beschäftigt, daß er den Geschwindigkeitsgewinn ausbremst. Sinnvollerweise teilt der zentrale Prozeß das Bild in Streifen ein und schickt deren Eckkoordinaten an die 'Arbeiter'. Auch hier ist es effizienter, zusammenhängende Blöcke zu verschicken.

Leicht ist bei diesem Beispiel auch eine Fehlertoleranz herzustellen. Der Master könnte zunächst Protokoll darüber führen, welche Aufträge die Worker erhalten haben. Sollte ein Worker ausfallen oder blockiert sein und deshalb in vernünftiger Zeit kein Resultat melden, gibt der Master dessen Auftrag einem weiteren Worker, der keine Berechnung mehr auszuführen hat. In der Praxis laufen Programme eher Stunden und Tage als Sekunden, so daß Ausfälle durchaus wahrscheinlich sind. Werden die Rechner des Clusters auch anderweitig genutzt, kommt es obendrein zu Lastschwankungen. Hier könnte ein Verteilungsprotokoll ansetzen.

Im Standardumfang des PVM-Pakets findet sich auch eine grafische Version der Apfelmännchen-Berechnung namens xep. Die fertigen Bildstreifen landen sofort auf dem Monitor: Der Master frischt die Anzeige auf, sobald Worker neue Datenpakete schicken.

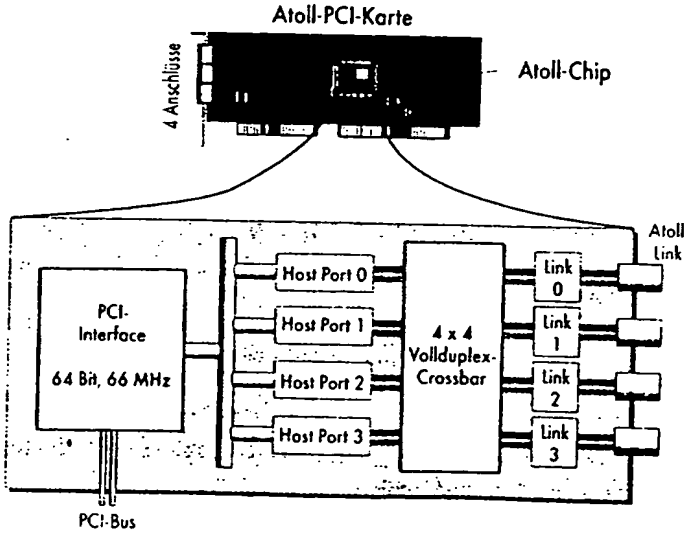
Haben die Rechner des Clusters Kontakt (Remote-Login mit rsh oder ssh muß funktionieren), läßt sich xep schnell in Betrieb nehmen. Man erzeugt eine Textdatei namens hostfile, in welcher Zeile für Zeile die IP-Nummern der Cluster-Rechner aufgelistet sind. Dann ruft man auf dem zentralen Rechner

```
pvm hostfile
```

Daten-Eilpost

Die Bandbreite des Systembusses begrenzt die sinnvolle Anzahl an Prozessoren pro Einzelrechner auf acht, allerhöchstens 16. Werden weitere Prozessoren benötigt, muß man mehrere Rechner zum Cluster verknüpfen. Problematisch wird dann die Kommunikation der Systeme: die Allerweltstechnik des (Fast) Ethernet gerät bei einer massiv parallelen Applikation schnell an ihre Grenzen; sowohl die Zeit, bis eine Nachricht eintrifft (Latenzzeit), als auch die maximale Bandbreite sinken der Leistung heutiger Prozessoren um Größenordnungen hinterher.

Beispiel: Müssen zunächst viele Kontrollnachrichten verschickt werden, die anschließend den eigentlichen Datenaustausch initiieren, kann schon der Aufruf der Sende- und Empfangsroutinen des Betriebssystems mehr Zeit kosten als die lokale Berechnung von vorhandenen Teilproblemen. Vermeidet man jedoch den Austausch von Kontrollnachrichten, kann die parallele Ausführung leiden – indem Prozessoren nach getaner Arbeit ohne neue Aufgaben bleiben und damit die Gesamtleistung verschlechtern.



Die an der Universität Mannheim entwickelte Atoll-Karte (Hardwarekosten ca. 1000 Mark) teilt jedem Hauptprozessor eines Vierfach-Boards eines der vier Devices zu. [5]

Daher geht der Trend dahin, Cluster mit schnellen Verbindungen zu betreiben. Im Vergleich zu Fast Ethernet – mit seinen Latenzzeiten von mehreren hundert µsec – verkürzen Produkte wie Myrinet, SCI oder Atoll diese Zeiten auf etwa 10 µsec. Die 'User Level Communication' umgeht das Betriebssystem, spart damit dessen zeitraubende Funktionsaufrufe und vergrößert die Bandbreite. Eingeschränkt nur noch durch die Leistung der PCI-Bridge, erreicht man Werte bis zu 100 MByte/s. Spezielle PVM-Versionen sind für derartige Netzwerke optimiert worden.

auf und verläßt pvm gleich wieder mit quit. Der Befehl xep & startet nun die PVM-Dämonen und die Berechnung: Nur wenige Tastendrücke für den Benutzer, aber ein großer Schritt für die Rechenleistung. (jl)

Literatur

[1] Ian Foster. Designing and building parallel programs. Addison Wesley. 1995. www-unix.mcs.anl.gov/dbpp/

[2] Al Geist u. a.. PVM – a users' guide and tutorial for networked parallel computing. MIT Press. 1994. www.netlib.org/pvm3/book/pvm-book.html

[3] PVM-Webseite. www.epm.ornl.gov/pvm/pvm_home.html

[4] Arne Schäpers. Why don't we do it in a row?. DCOM als Alternative zum Server-Clustering. c't 6/98. S. 390

[5] Atoll-Netzwerkinterface. www.atoll-net.de


```
#include <math.h>
#include "pvm3.h"

int main(int argc, char **argv)
{
    int mytid;              /* Task-ID */
    int mastertid;          /* ID des Auftraggebers */
    double x1, y1, x2, y2; /* Koordinaten des Streifens*/
    int wd, ht;             /* Größe des Streifens */
    char *pix;              /* Berechnetes Bild */
    int mid;
    mytid = pvm_mytid();
    /* auf Anfrage warten, Größe und Koordinaten empfangen */
    while ((mid = pvm_recv(-1, 1)) > 0) {
        pvm_bufinfo(mid, (int*)0, (int*)0, &mastertid);
        /* entpacke Information */
        pvm_unpackf("%lf %lf %lf %lf %d %d", &x1, &y1, &x2, &y2, &wd, &ht);
        /* berechne das Bild */
        pix = calc_tile(x1, y1, x2, y2, wd, ht);
        /* packe berechnete Daten */
        pvm_packf("%* %c", PvmDataDefault, wd * ht, pix);
        /* sende Bilddaten an Master */
        if (pvm_send(mastertid, 2))
            fprintf(stderr, "Fehler beim Senden.\n");
        free(pix);
    }
    pvm_exit();
    exit(1);
}

/* eigentliche Mandelbrot-Rechnung */

char* calc_tile(double x1, double y1,double x2,double y2,int wd,int ht)
{
    char *pix;              /* berechnetes Bild */
    int ix, iy;             /* aktuelles Pixel */
    double x, y;            /* Realteil, Imaginärteil */
    register double ar, ai; /* Akkumulator */
    register double a1, a2;
    register int ite;       /* aktuelle Zahl der Iterationen */

    if (wd < 1 || wd > 2048 || ht < 1 || ht > 2048) {
        fprintf(stderr, "Falsche Breite/Höhe\n");
        pvm_exit();
        exit(1);
    }
    pix = (char*)malloc(wd * ht);
    x2 -= x1;
    y2 -= y1;
    for (iy = ht; iy-- > 0; ) {
        y = (iy * y2) / ht + y1;
        for (ix = wd; ix-- > 0; ) {
            x = (ix * x2) / wd + x1;
            ar = x;
            ai = y;
            for (ite = 0; ite < 255; ite++) {
                a1 = (ar * ar);
                a2 = (ai * ai);
                if (a1 + a2 > 4.0)
                    break;
                ai = 2 * ai * ar + y;
                ar = a1 - a2 + x;
            }
            pix[iy * wd + ix] = ~ite;
        }
    }
    return pix;
}
```

Der Arbeiter 'mtile' wartet auf den Auftrag, nimmt entgegen, welchen Bildstreifen er berechnen soll, und sendet sein Ergebnis zurück.

```
#include <stdio.h>
#include "pvm3.h"

char *mandelbrot();
int nprocessors = 1;        /* enthält später Zahl der Prozessoren */
int *prtids = 0;            /* zeigt später auf Array mit Task-IDs */

int main(int argc, char **argv) { /* Parameter: Breite Höhe x1 y1 x2 y2 */
    int mytid;              /* meine Task-ID*/
    int wd, ht;             /* Bildgröße */
    double x1, y1, x2, y2;  /* Eckpunkte */
    char *pix;              /* Bilddaten */
    int i;                  /* Index für Prozeß */
    wd = atoi(argv[1]);     ht = atoi(argv[2]);
    x1 = atof(argv[3]);     y1 = atof(argv[4]);
    x2 = atof(argv[5]);     y2 = atof(argv[6]);
    /* Anmeldung bei PVM */
    if ((mytid = pvm_mytid()) < 0)  exit(1);
    /* Ermittle die Anzahl der Prozessoren der Clusters */
    pvm_config(&nprocessors, (int*)0, (struct pvmhostinfo**)0);
    fprintf(stderr, "%d tile Worker benutzt.\n", nprocessors);
    /* Starten der Worker-Tasks */
    prtids = (int*)malloc(nprocessors * sizeof(int));
    for (i = 0; i < nprocessors; i++)
        if (pvm_spawn("mtile", (char**)0, 0, "", 1, &prtids[i]) < 0)
            fprintf(stderr, "Kann Worker nicht starten.\n");
    pix = mandelbrot(x1, y1, x2, y2, wd, ht);
    /* Beende Worker-Prozesse */
    for (i = 0; i < nprocessors; i++)  pvm_kill(prtids[i]);
    /* Abmelden von PVM */
    pvm_exit();  exit(0);
}

char* mandelbrot(double x1, double y1, double x2, double y2, int wd,int ht) {
    char *pix = 0;      /* berechnetes Bild */
    char *tile = 0;     /* Daten eines Streifens */
    int *tpos;          /* Position eines Streifens */
    int maxwd = 0;      /* maximale Breite eines Streifens */
    int slavetid;       /* Prozessor-ID */
    int twd;            /* Breite */
    double xyxy[4];  int wdht[2];
    int i, j;  int y;  char *ba1, *ba2;
    pix = (char*)malloc(wd * ht);
    /* Aufteilung in Streifen und Zuweisung an Prozessoren */
    tpos = (int*)malloc((nprocessors + 1) * sizeof(int));
    x2 -= x1;   xyxy[0] = x1;   xyxy[1] = y1;   xyxy[3] = y2;
    wdht[1] = ht;   tpos[0] = 0;
    fprintf(stderr,"Sende Aufträge an Prozessoren.\n");
    for (i = 0; i < nprocessors; i++) {
        /* bereite Puffer vor */
        pvm_initsend(PvmDataDefault);
        tpos[i + 1] = ((i + 1) * wd) / nprocessors;
        wdht[0] = tpos[i + 1] - tpos[i];
        if (wdht[0] > maxwd)   maxwd = wdht[0];
        xyxy[2] = (tpos[i + 1] * x2) / wd + x1;
        /* verpacke Bilddaten */
        pvm_pkdouble(xyxy, 4, 1);  pvm_pkint(wdht, 2, 1);
        /* sende Daten an Prozeß */
        if (pvm_send(prtids[i], 1)) {
            fprintf(stderr, "Fehler beim Senden an <%x>\n", prtids[i]);
            pvm_exit();  exit(1);
        }
        xyxy[0] = xyxy[2];
    }
    /* Empfange Daten und füge sie zusammen */
    tile = (char*)malloc(maxwd * ht);
    fprintf(stderr,"Prozessoren antworten:");
    for (i = 0; i < nprocessors; i++) {
        /* empfange von allen Prozessen (-1 = Wildcard) */
        if (pvm_recv(-1, 2) < 1) {
            fprintf(stderr, "Fehler beim Empfang.\n");
            pvm_exit();  exit(1);
        }
        /* untersuche eingegangene Nachrichten */
        pvm_bufinfo(pvm_getrbuf(), &j, (int*)0, &slavetid);
        for (j = 0; j < nprocessors; j++)
            if (prtids[j] == slavetid) break;
        if (j < nprocessors) {
            fprintf(stderr, " %d", j);  fflush(stderr);
            twd = tpos[j + 1] - tpos[j];
            /* entpacke Nachricht */
            pvm_upkbyte(tile, twd * ht, 1);
            ba1 = tile;  ba2 = pix + tpos[j];
            for (y = ht; y-- > 0; ) {
                BCOPY(ba1, ba2, twd);  ba1 += twd;  ba2 += wd;
            }
        }
    }
    free(tpos);  free(tile);
    return pix;
}
```

Der Master-Prozeß 'mmain' startet die Arbeitsprozesse und teilt ihnen mit, welche Streifen des Apfelmännchens (Mandelbrot-Menge) sie jeweils berechnen sollen. c't